

Design of Digital Circuits

Lecture 17a: Superscalar Execution

Prof. Onur Mutlu

ETH Zurich

Spring 2019

18 April 2019

Required Readings

■ This week

- ❑ Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proceedings of the IEEE, 1995
- ❑ H&H Chapters 7.8 and 7.9
- ❑ McFarling, “Combining Branch Predictors,” DEC WRL Technical Report, 1993.

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms

Recall: OOO Execution: Restricted Dataflow

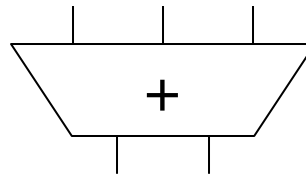
- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
 - which piece?
- The dataflow graph is limited to the **instruction window**
 - Instruction window: all decoded but not yet retired instructions
- Can we do it for the whole program?
- Why would we like to?
- In other words, how can we have a large instruction window?
- Can we do it efficiently with Tomasulo's algorithm?

Recall: State of RAT and RS in Cycle 7

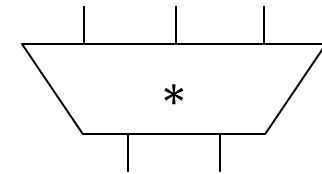
	Cycle	1	2	3	4	5	6	7
MUL R1, R2 → R3	F	D	E ₁	E ₂	E ₃	E ₄	E ₅	
ADD R3, R4 → R5		F	D	-	-	-	-	
ADD R2, R6 → R7			F	D	E ₁	E ₂	E ₃	
ADD R8, R9 → R10				F	D	E ₁	E ₂	
MUL R7, R10 → R11					F	D	-	
ADD R5, R11 → R5						F	D	

Register	Valid	Tag	Value
R1	1		1
R2	1		2
R3	0	x	
R4	1		4
R5	0	d	
R6	1		6
R7	0	b	
R8	1		8
R9	1		9
R10	0	c	
R11	0	y	

	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	



	Source 1			Source 2		
	V	Tag	Value	V	Tag	Value
x	1	~	1	1	~	2
y	0	b		0	c	
z						
t						



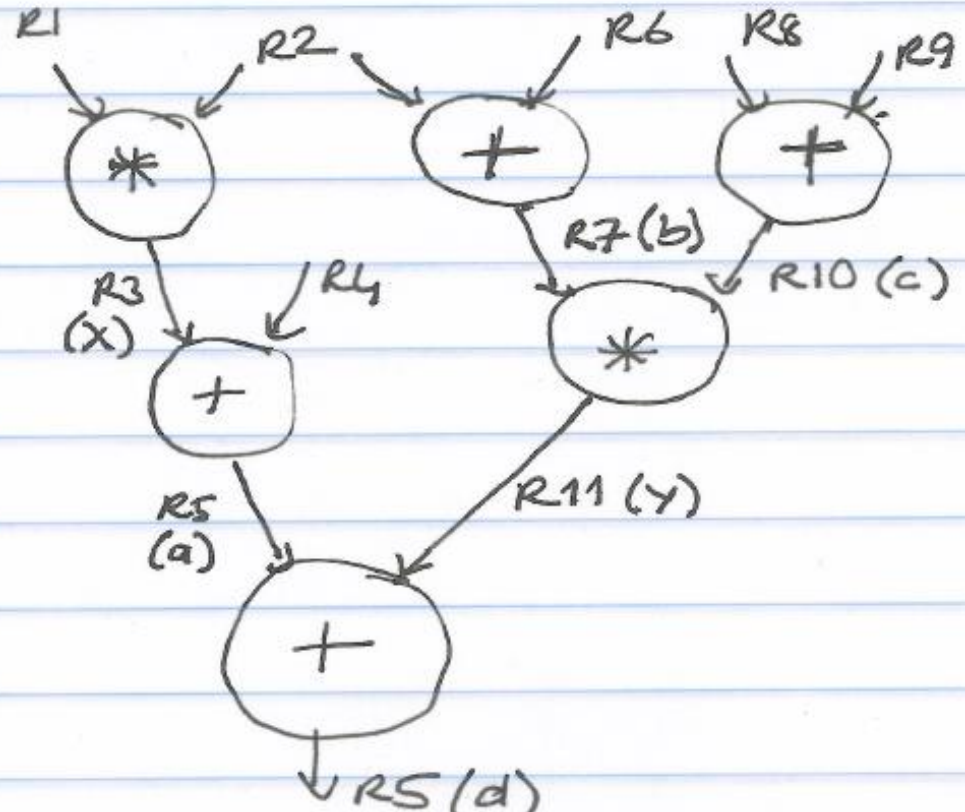
Recall: Dataflow Graph

MUL R1, R2 → R3 (x)
ADD R3, R4 → R5 (a)
ADD R2, R6 → R7 (b)
ADD R8, R9 → R10 (c)
MUL R7, R10 → R11 (y)
ADD R5, R11 → R5 (d)

Dataflow graph

Nodes: operations performed by the instruction

Arcs: tags in Tomasulo's algorithm



Other Approaches to Concurrency (or Instruction Level Parallelism)

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

Review: Data Flow: Exploiting Irregular Parallelism

Data Flow Summary

- Availability of data determines order of execution
- A data flow node fires when its sources are ready
- Programs represented as data flow graphs (of nodes)
- Data Flow at the ISA level has not been (as) successful
- Data Flow implementations at the microarchitecture level (while preserving Von Neumann semantics) have been very successful
 - Out of order execution is the prime example

Pure Data Flow Advantages/Disadvantages

■ Advantages

- ❑ Very good at exploiting **irregular parallelism**
- ❑ Only real dependencies constrain processing
- ❑ More parallelism can be exposed than Von Neumann model

■ Disadvantages

- ❑ No precise state semantics
 - Debugging very difficult
 - Interrupt/exception handling is difficult (what is precise state semantics?)
- ❑ Too much parallelism? (Parallelism control needed)
- ❑ High bookkeeping overhead (tag matching, data storage)
- ❑ ...

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

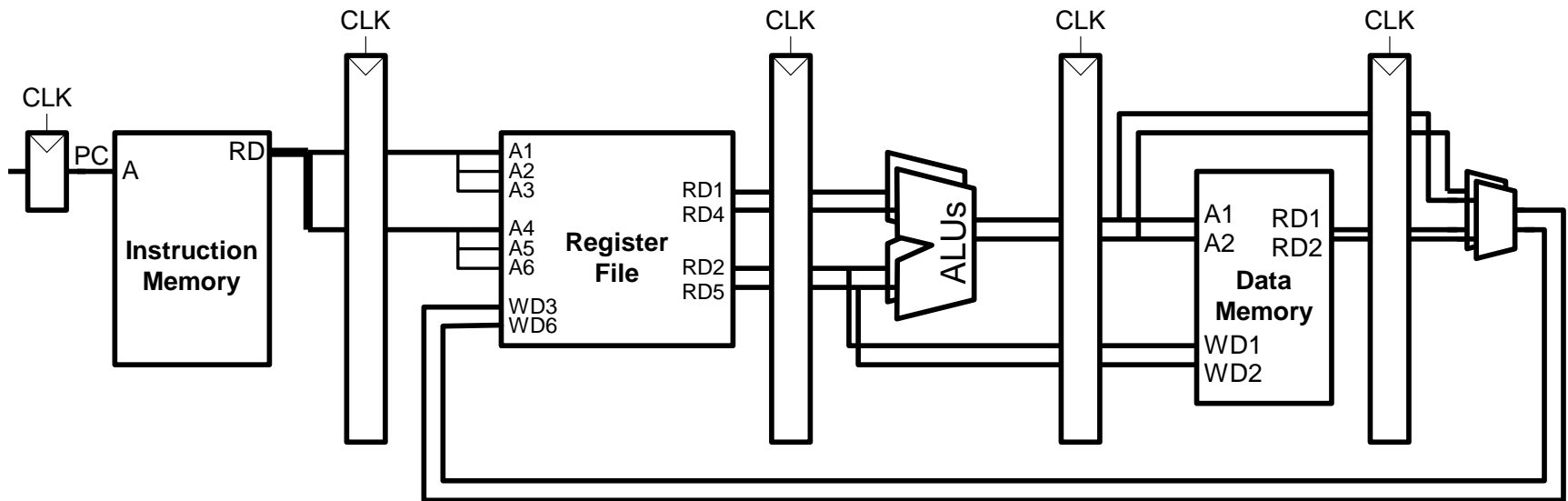
Superscalar Execution

Superscalar Execution

- Idea: Fetch, decode, execute, retire **multiple instructions per cycle**
 - N-wide superscalar → N instructions per cycle
- Need to add the hardware resources for doing so
- Hardware performs the dependence checking between concurrently-fetched instructions
- Superscalar execution and out-of-order execution are orthogonal concepts
 - Can have all four combinations of processors:
[in-order, out-of-order] x [scalar, superscalar]

In-Order Superscalar Processor Example

- Multiple copies of datapath: Can fetch/decode/execute multiple instructions per cycle
- Dependencies make it tricky to issue multiple instructions at once

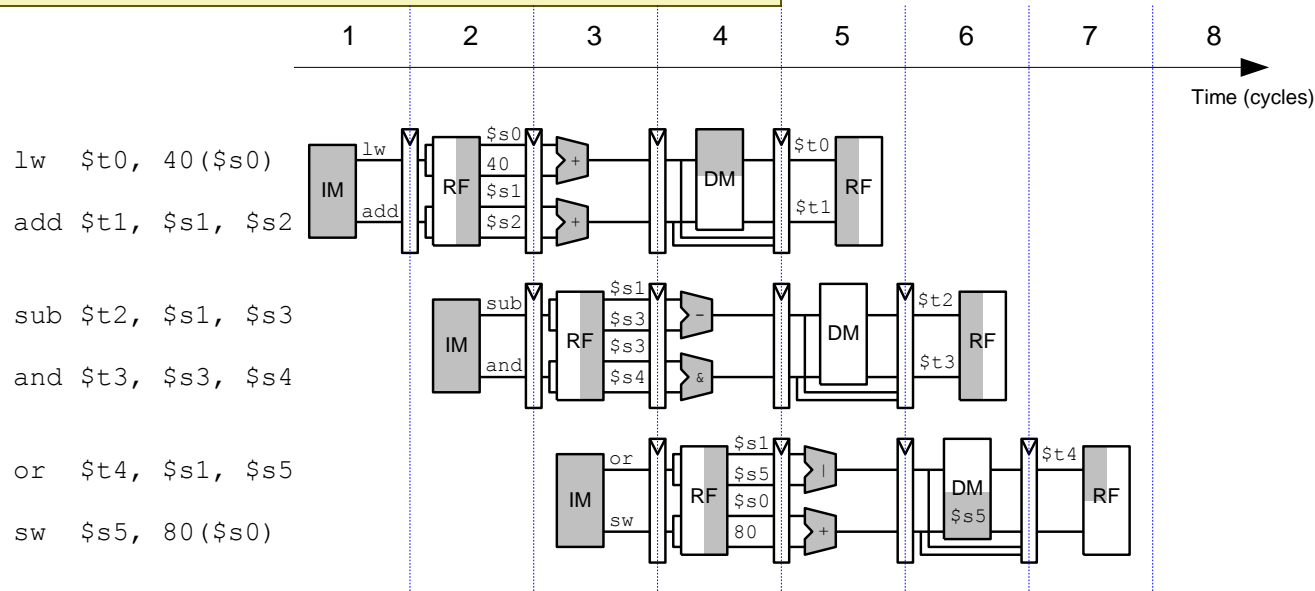


Here: Ideal IPC = 2

In-Order Superscalar Performance Example

```
lw  $t0, 40($s0)
add $t1, $s1, $s2
sub $t2, $s1, $s3
and $t3, $s3, $s4
or  $t4, $s1, $s5
sw  $s5, 80($s0)
```

Ideal IPC = 2

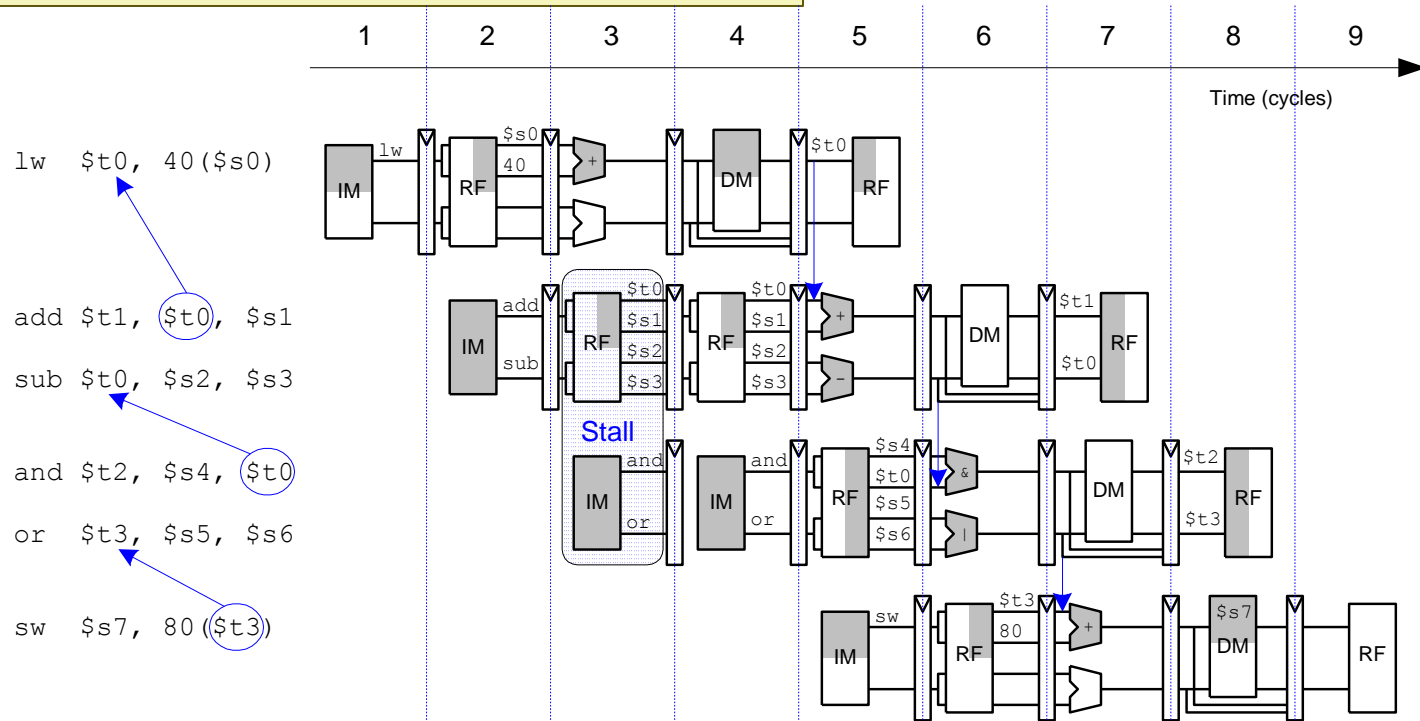


Actual IPC = 2 (6 instructions issued in 3 cycles)

Superscalar Performance with Dependencies

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Ideal IPC = 2



Actual IPC = 1.2 (6 instructions issued in 5 cycles)

Superscalar Execution Tradeoffs

■ Advantages

- Higher IPC (instructions per cycle)

■ Disadvantages

- Higher complexity for dependency checking
 - Require checking within a pipeline stage
 - Renaming becomes more complex in an OoO processor
- More hardware resources needed

Design of Digital Circuits

Lecture 17a: Superscalar Execution

Prof. Onur Mutlu

ETH Zurich

Spring 2019

18 April 2019

Approaches to (Instruction-Level) Concurrency

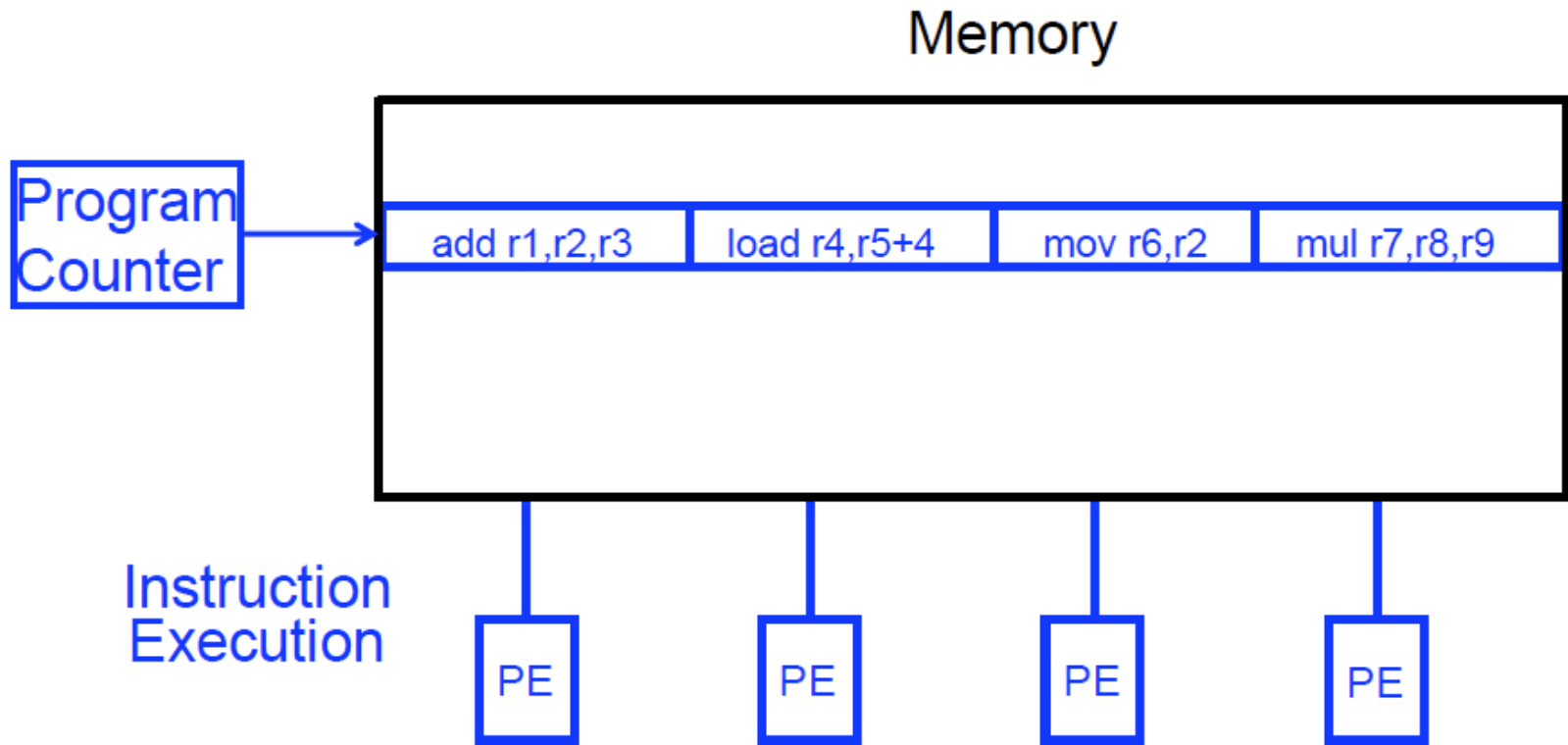
- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

VLIW

VLIW Concept

- Superscalar
 - **Hardware** fetches multiple instructions and checks dependencies between them
- VLIW (Very Long Instruction Word)
 - **Software (compiler) packs independent instructions** in a larger “instruction bundle” to be fetched and executed concurrently
 - Hardware fetches and executes the instructions in the bundle concurrently
- No need for hardware dependency checking between concurrently-fetched instructions in the VLIW model

VLIW Concept



- Fisher, “**Very Long Instruction Word architectures and the ELI-512,**” ISCA 1983.
 - ELI: Enormously longword instructions (512 bits)

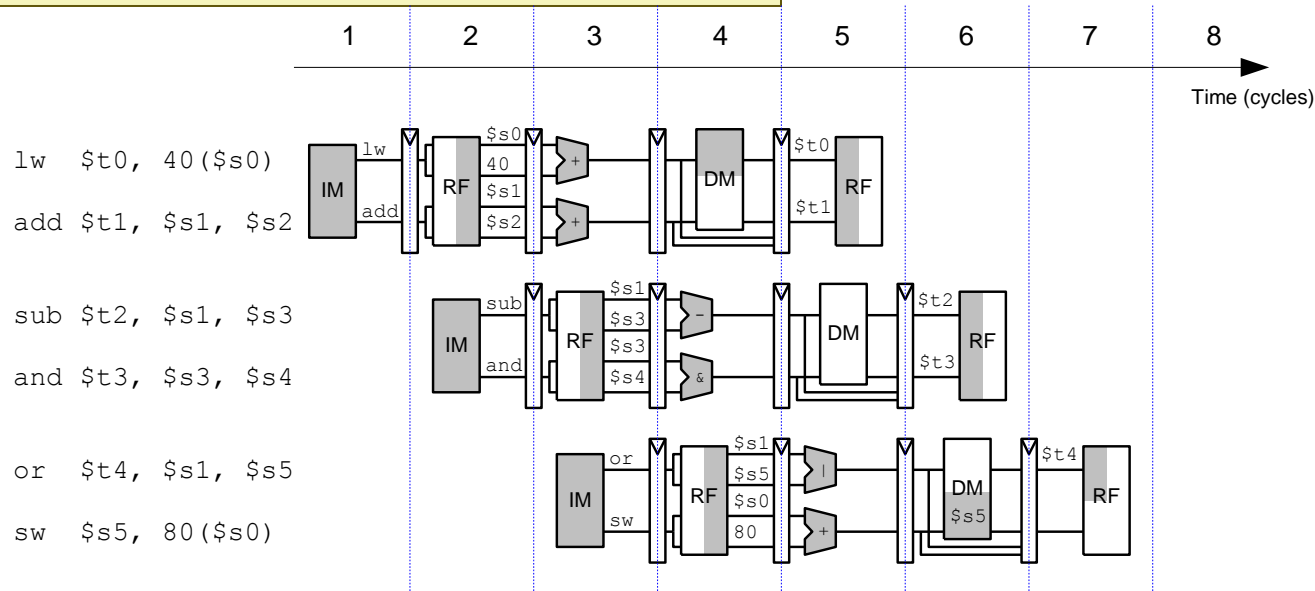
VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
 - Packed instructions can be logically unrelated (contrast with SIMD/vector processors, which we will see soon)
- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction
- Traditional Characteristics
 - Multiple functional units
 - All instructions in a bundle are executed in lock step
 - Instructions in a bundle statically aligned to be directly fed into the functional units

VLIW Performance Example (2-wide bundles)

```
lw  $t0, 40($s0)
add $t1, $s1, $s2
sub $t2, $s1, $s3
and $t3, $s3, $s4
or  $t4, $s1, $s5
sw  $s5, 80($s0)
```

Ideal IPC = 2



Actual IPC = 2 (6 instructions issued in 3 cycles)

VLIW Lock-Step Execution

- Lock-step (all or none) execution: If any operation in a VLIW instruction stalls, all instructions stall
- In a truly VLIW machine, the compiler handles all dependency-related stalls, hardware does **not** perform dependency checking
 - What about variable latency operations?

VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
 - Except multiple instructions in parallel
- RISC (John Cocke, 1970s, IBM 801 minicomputer)
 - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
 - And, to reorder simple instructions for high performance
 - Hardware does little translation/decoding → very simple
- VLIW (Josh Fisher, ISCA 1983)
 - Compiler does the hard work to find instruction level parallelism
 - Hardware stays as simple and streamlined as possible
 - Executes each instruction in a bundle in lock step
 - Simple → higher frequency, easier to design

Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
 - Most successful commercially
- Intel IA-64
 - Not fully VLIW, but based on VLIW principles
 - EPIC (Explicitly Parallel Instruction Computing)
 - Instruction bundles can have dependent instructions
 - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

VLIW Tradeoffs

■ Advantages

- + No need for dynamic scheduling hardware → simple hardware
- + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
- + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

■ Disadvantages

- Compiler needs to find N independent operations per cycle
 - If it cannot, inserts NOPs in a VLIW instruction
 - Parallelism loss AND code size increase
- Recompile required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
- Lockstep execution causes independent operations to stall
 - No instruction can progress until the longest-latency instruction completes

VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
 - Solely-compiler approach of VLIW has several downsides that reduce performance
 - Too many NOPs (not enough parallelism discovered)
 - Static schedule intimately tied to microarchitecture
 - Code optimized for one generation performs poorly for next
 - No tolerance for variable or long-latency operations (lock step)
- ++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
- Enable code optimizations
- ++ VLIW successful when parallelism is easier to find by the compiler (traditionally embedded markets, DSPs)

An Example Work: Superblock

The Superblock: An Effective Technique for VLIW and Superscalar Compilation

Wen-mei W. Hwu Scott A. Mahlke William Y. Chen Pohua P. Chang

Nancy J. Warter Roger A. Bringmann Roland G. Ouellette Richard E. Hank

Tokuzo Kiyohara Grant E. Haab John G. Holm Daniel M. Lavery *

Hwu et al., [The superblock: An effective technique for VLIW and superscalar compilation.](#)
The Journal of Supercomputing, 1993.

- **Lecture Video on Static Instruction Scheduling**
 - <https://www.youtube.com/watch?v=isBEVkIjgGA>

Another Example Work: IMPACT

IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors

Pohua P. Chang

Scott A. Mahlke

William Y. Chen

Nancy J. Warter

Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

The performance of multiple-instruction-issue processors can be severely limited by the compiler's ability to generate efficient code for concurrent hardware. In the IMPACT project, we have developed IMPACT-I, a highly optimizing C compiler to exploit instruction level concurrency. The optimization capabilities of the IMPACT-I C compiler are summarized in this paper. Using the IMPACT-I C compiler, we ran experiments to analyze the performance of multiple-instruction-issue processors executing some important non-numerical programs. The multiple-instruction-issue processors achieve solid speedup over high-performance single-instruction-issue processors.

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

Recall: How to Handle Data Dependences

- Anti and output dependences are easier to handle
 - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
 - Detect and wait until value is available in register file
 - Detect and forward/bypass data to dependent instruction
 - Detect and eliminate the dependence at the software level
 - No need for the hardware to detect dependence
 - Predict the needed value(s), execute “speculatively”, and verify
 - Do something else (fine-grained multithreading)
 - No need to detect

How to Handle Control Dependences

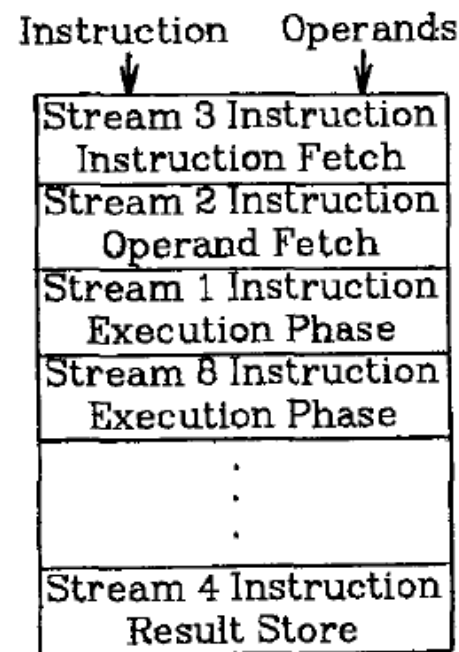
- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- **Potential solutions if the instruction is a control-flow instruction:**
 - Stall the pipeline until we know the next fetch address
 - Guess the next fetch address (branch prediction)
 - Employ delayed branching (branch delay slot)
 - Do something else (fine-grained multithreading)
 - Eliminate control-flow instructions (predicated execution)
 - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Fine-Grained Multithreading

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
 - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
 - Branch/instruction resolution latency overlapped with execution of other threads' instructions

- + No logic needed for handling control and data dependences within a thread
- Single thread performance suffers
- Extra logic for keeping thread contexts
- Does not overlap latency if not enough threads to cover the whole pipeline



Fine-Grained Multithreading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently
- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads
- Thornton, “Parallel Operation in the Control Data 6600,” AFIPS 1964.
- Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.

Fine-Grained Multithreading: History

- CDC 6600's peripheral processing unit is fine-grained multithreaded
 - Thornton, “[Parallel Operation in the Control Data 6600](#),” AFIPS 1964.
 - Processor executes a different I/O thread every cycle
 - An operation from the same thread is executed every 10 cycles

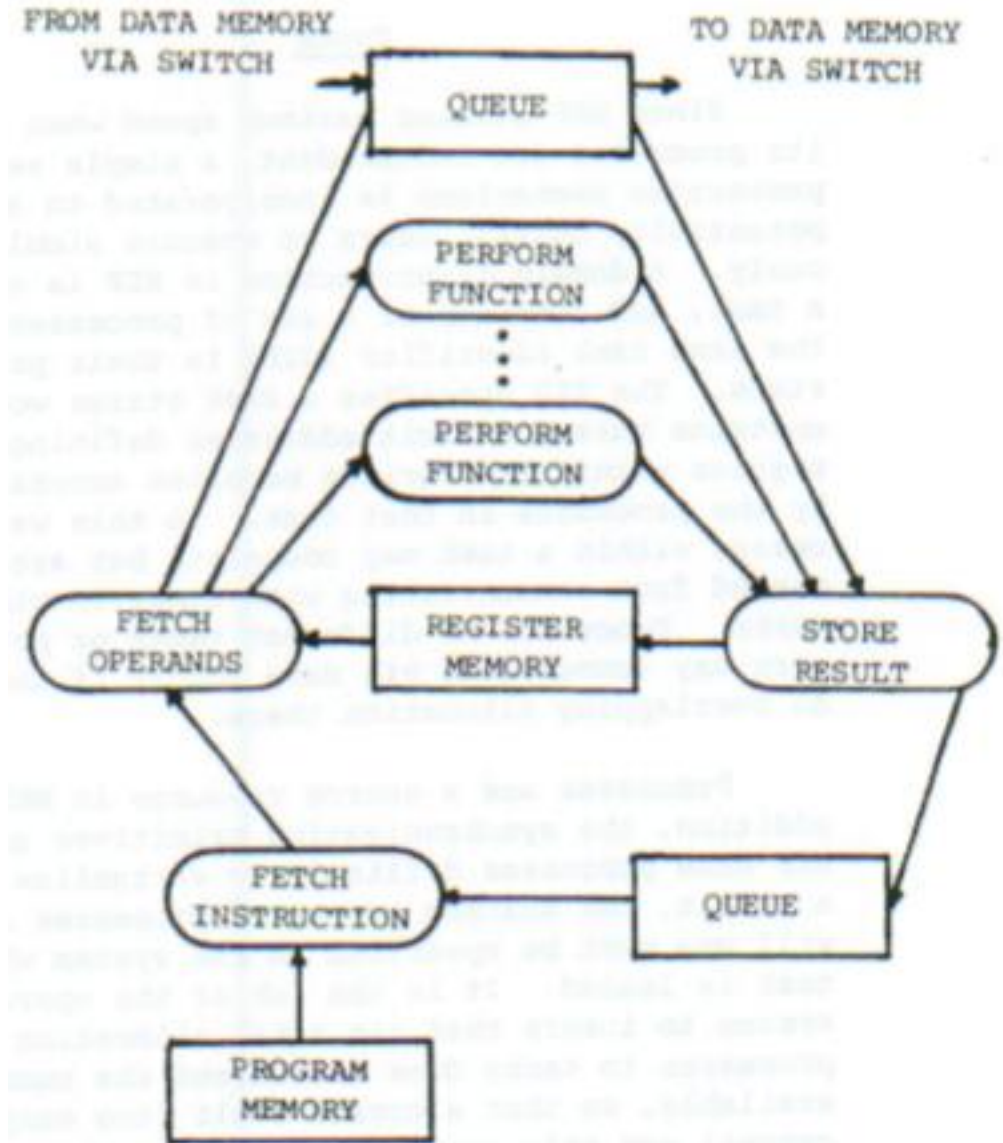
- Denelcor HEP (Heterogeneous Element Processor)
 - Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978.
 - 120 threads/processor
 - available queue vs. unavailable (waiting) queue for threads
 - each thread can have only 1 instruction in the processor pipeline; each thread independent
 - to each thread, processor looks like a non-pipelined machine
 - system throughput vs. single thread performance tradeoff

Fine-Grained Multithreading in HEP

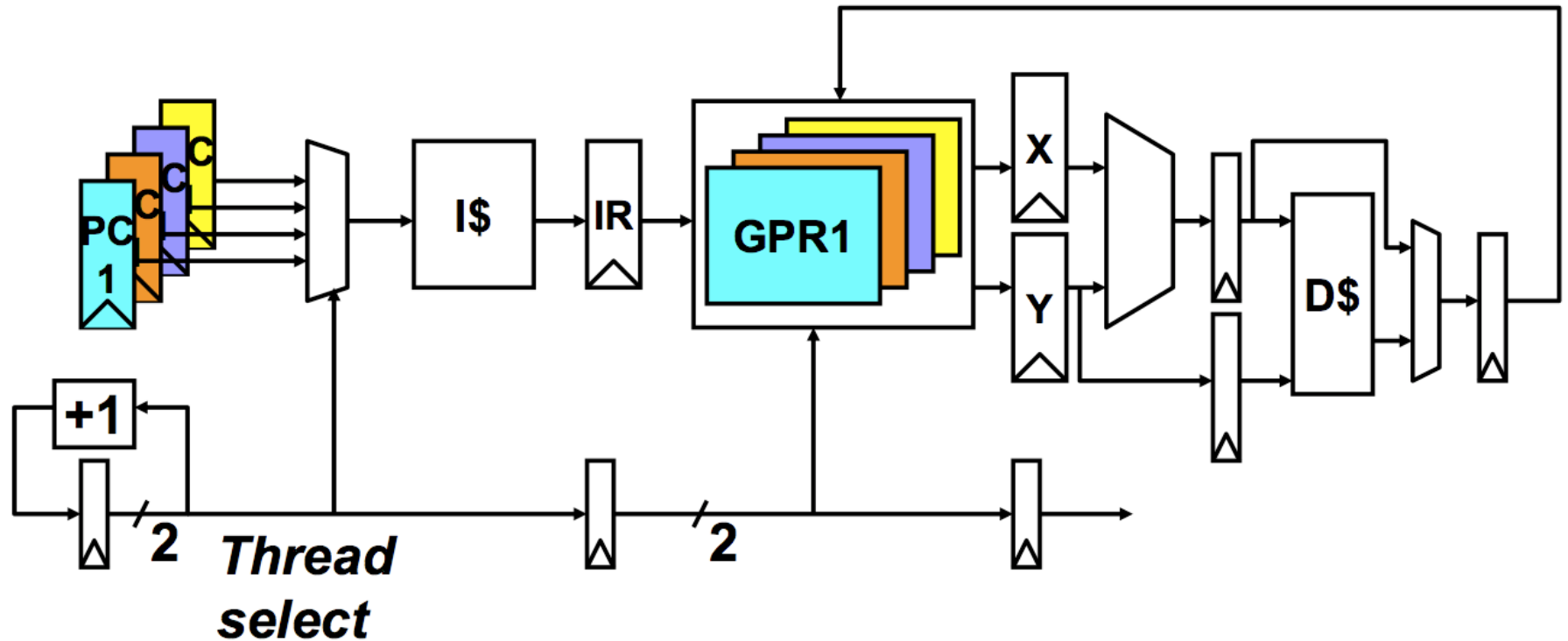
- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
 - ❑ assuming no memory access
- No control and data dependency checking



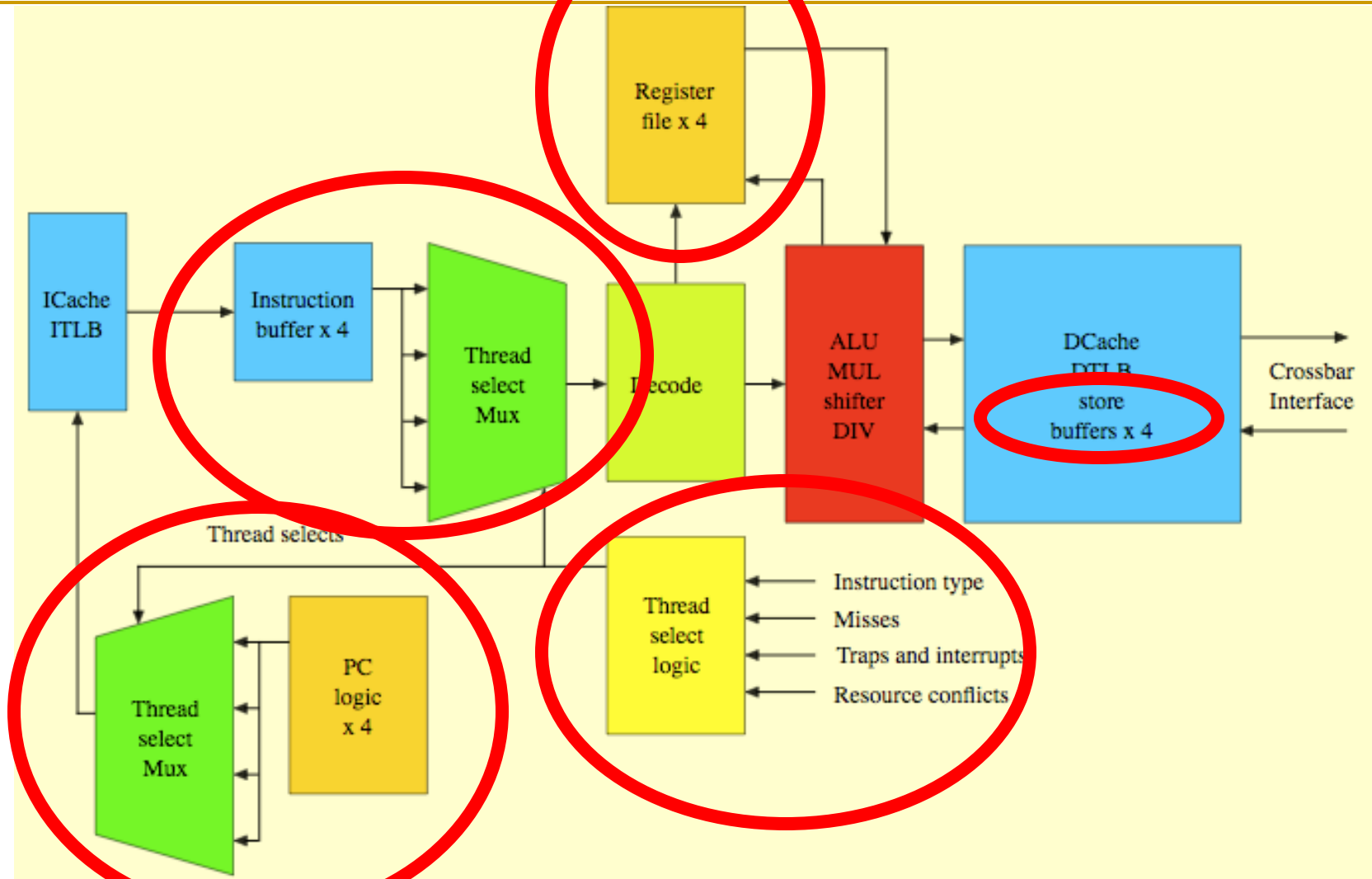
Burton Smith
(1941-2018)



Multithreaded Pipeline Example



Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

Fine-grained Multithreading

■ Advantages

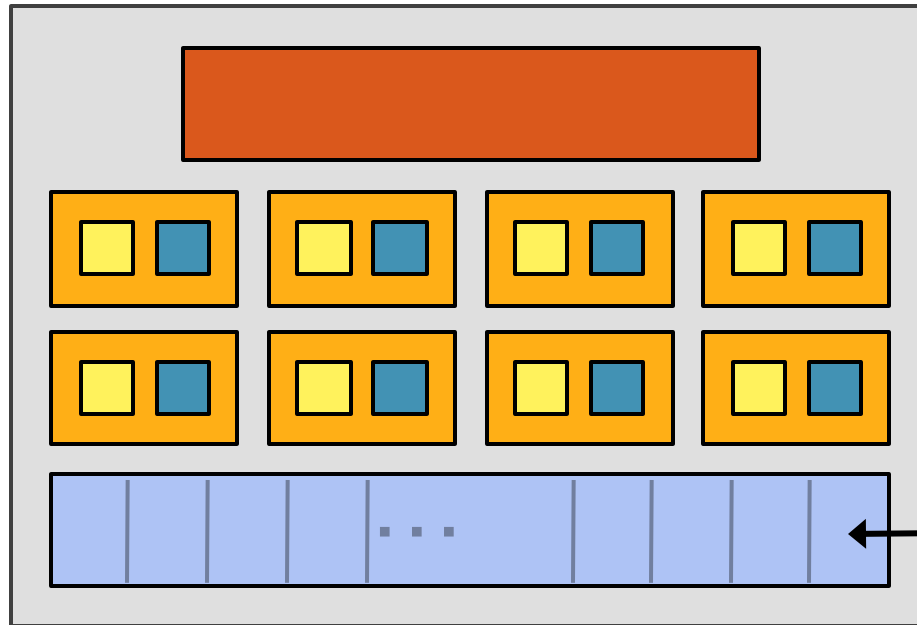
- + No need for dependency checking between instructions
(only one instruction in pipeline from a single thread)
- + No need for branch prediction logic
- + Otherwise-bubble cycles used for executing useful instructions from different threads
- + Improved system throughput, latency tolerance, utilization

■ Disadvantages

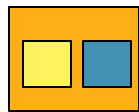
- Extra hardware complexity: multiple hardware contexts (PCs, register files, ...), thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles from the same thread)
- Resource contention between threads in caches and memory
- Some dependency checking logic *between* threads remains (load/store)

Modern GPUs are FGMT Machines

NVIDIA GeForce GTX 285 “core”



64 KB of storage
for thread contexts
(registers)



= data-parallel (SIMD) func. unit,
control shared across 8 units

■ = multiply-add
■ = multiply

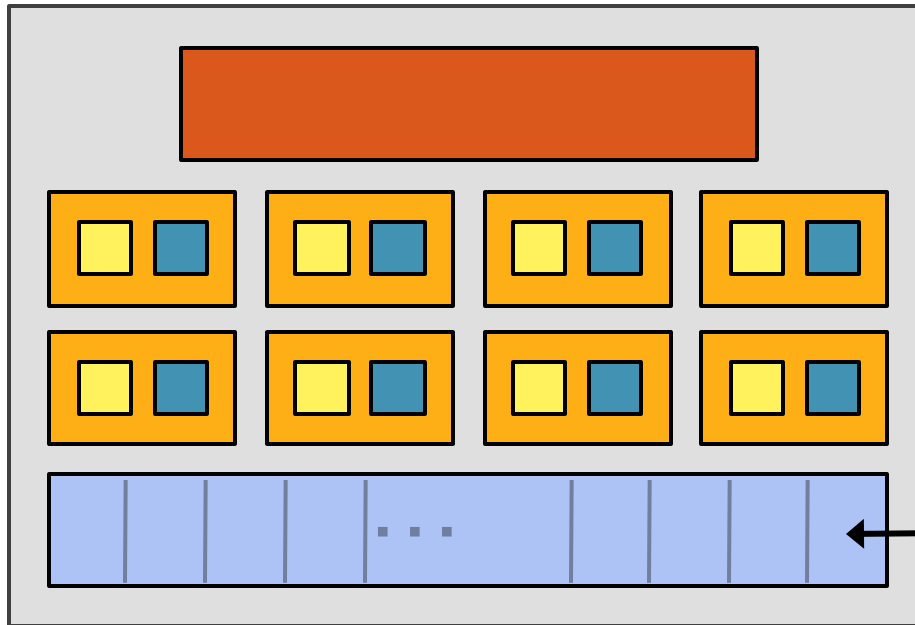


= instruction stream decode



= execution context storage

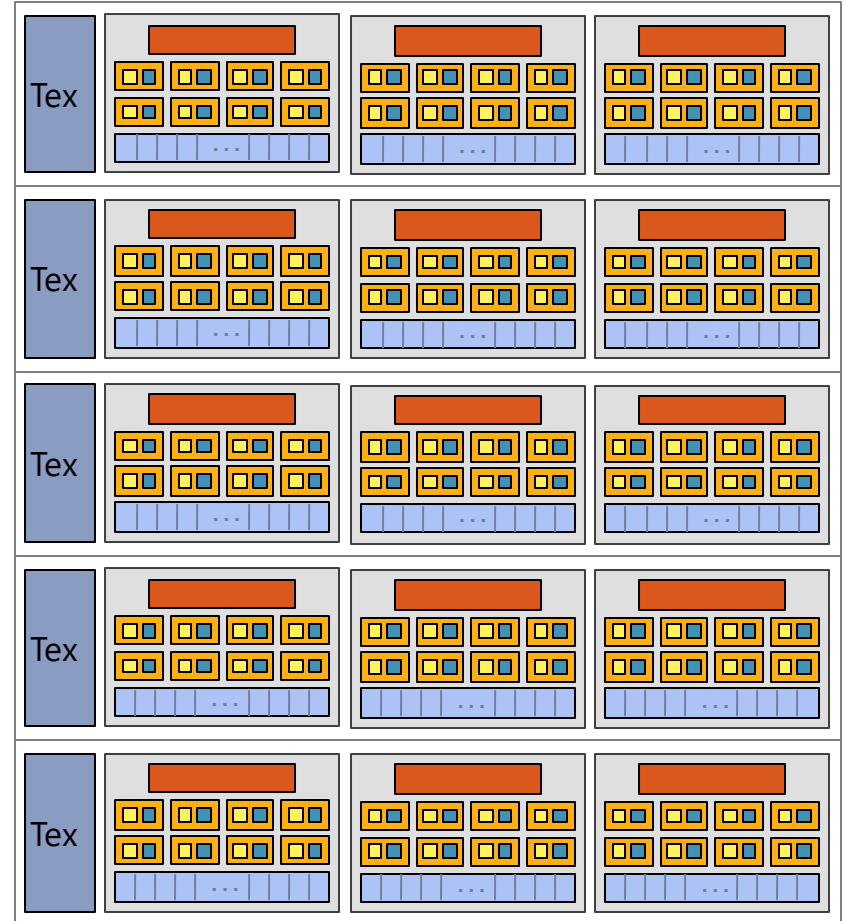
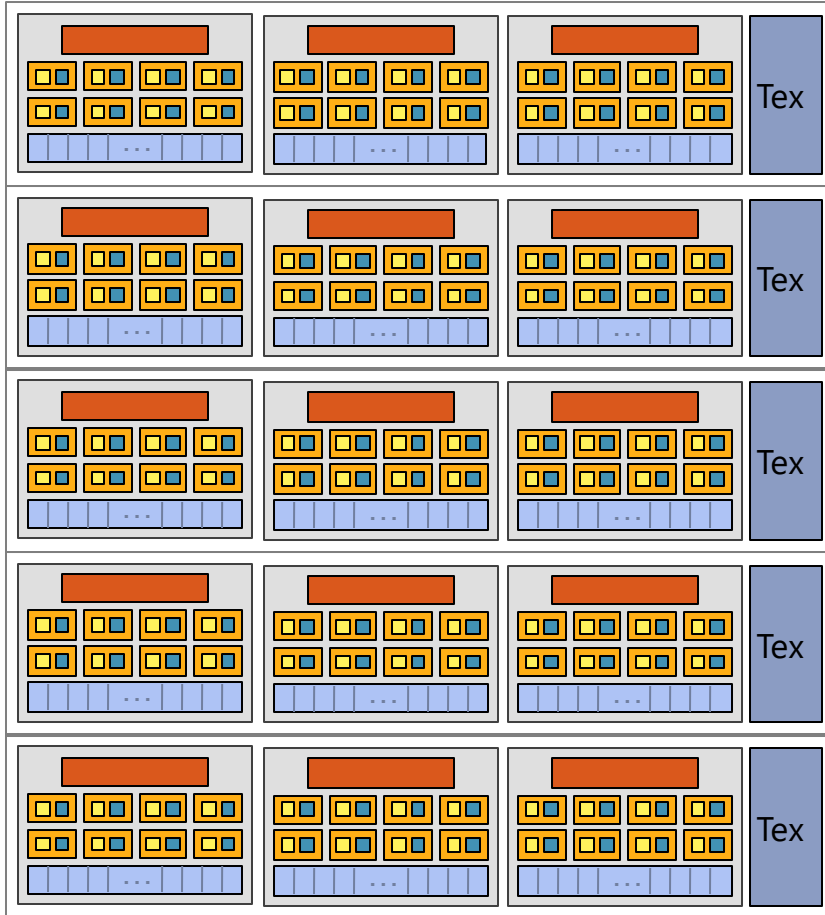
NVIDIA GeForce GTX 285 “core”



64 KB of storage
for thread contexts
(registers)

- Groups of 32 **threads** share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

End of Fine-Grained Multithreading